VNUHCM–UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY
SUBJECT: **OBJECT–ORIENTED PROGRAMMING**

PROJECT

TOWER DEFENSE

TP.HCM, 20th August 2024

TABLE OF CONTENTS

1 Introduction

In this project we will combine techniques, basic data structures as well as object-oriented knowledge to build a simple game, tower defense.

To carry out this project, we need basic knowledge such as: file handling, processes, basic data structures and object-oriented knowledge...

The tutorial helps students build games at a basic level, and groups research on their own to complete it in the best way possible.

2 Screenplay

At the beginning of the game, a map and **a table of parameters** will appear that need the player to determine. Once all parameters are available, the game operates automatically, and players wait to see the results.

When all the "characters" are destroyed or when one of the characters has reached the finish line, the program will notify the player to select the 'y' key if they want to continue (the program will reset to the original state) or select 'any key' if you want to exit the game.

When you destroy all the characters, you advance to a new level (for example, the number of characters is larger, or the character can resist...). When you reach a certain level, it is considered a victory or the game restarts as at the beginning.

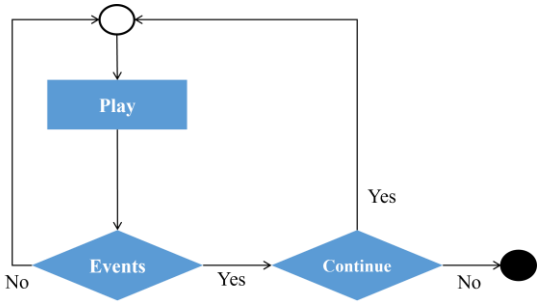

Figure 1: Diagram of screenplay

3 Steps to build a game

In this section we will go through the steps/classes of game building one by one. Note that this is just a programming suggestion, the student group designs their own appropriate templates during the project process.

Step 1: Because we MUST build in *Visual Studio*, the instructions are mainly in this programming environment. In this step, we will build a class named `cpoint` consisting of 2 files `cpoint.h` and `cpoint.cpp` as below.

```
// cpoint.h
class cpoint {
  public:
    // Constants and data
    const static int MAP_SIZE = 10;//  Map matrix size
    const static int LEFT = 3;//  Left-coordinates of the map screen
    const static int TOP = 1;// Top-coordinates of map screen
  private:
    int x, y, c;
  public:
    cpoint() { x = y = c = 0; }
    cpoint(int tx, int ty, int tc) { x = tx;y = ty;c = tc; }
    int getX() { return x; }
    int getY() { return y; }
    int getC() { return c; }
    void setC(int tc) { c = tc; }
    static cpoint fromXYToRowCol(cpoint v);
    static cpoint fromRowColToXY(cpoint s);
};
```

```
// cpoint.cpp
#include "cpoint.h"

cpoint cpoint::fromXYToRowCol(cpoint v) {
  return {(v.y - 1 - TOP)/2, (v.x - 2 - LEFT)/4, v.c };
}
cpoint cpoint::fromRowColToXY(cpoint s) {
  return { 4 * s.y + 2 + LEFT, 2 * s.x + 1 + TOP, s.c };
}
```

Code explanation: let's look at the `cpoint.h` file and see that there are constants `MAP_SIZE` which is the map size, here it is simply a matrix of 10 rows and 10 columns, the coordinates of the top (`TOP`) and left (`LEFT`) corners respectively, is 3 and 1. Of course this is just an illustration, the student group can adjust according to their needs. The figure below briefly describes these constants

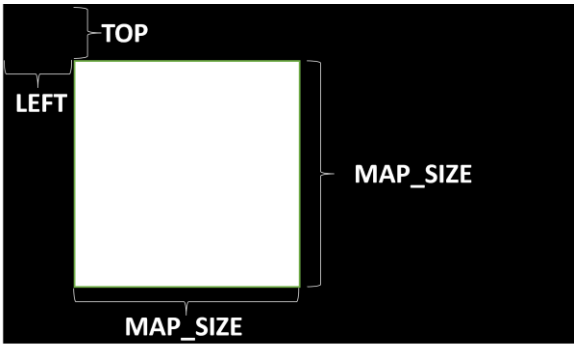

Figure 2: Game main screen

The properties in the `cpoint.h` file include x coordinates, y coordinates, and c is an additional property that depends on the context for further use. In this `cpoint` class we have 2 convenient–functions "`fromXYToRowCol`" and "`fromRowColToXY`" to facilitate conversion during processing. Consider for example that we have a point $P = \{5, 6, 0\}$, so point P has coordinates $x = 5$ and coordinates $y = 6$, and value $c = 0$. If we pass P to the method "`fromXYToRowCol`" then the result $P = \{2, 0, 0\}$. This means that now $x = 2$ (line number 2) and $y = 0$ (column number 0). On the contrary, if we now pass P to the "`fromRowColToXY`" method, the result $P = \{5, 6, 0\}$ is the same. Why is that? Simply, when we need to draw, we need the true coordinates of the screen, and when we calculate in a matrix, we need row and column information. Therefore, it is necessary to have methods to convert back and forth when needed. Of course this is just an illustration, the student group can think of a more convenient way. As for the `cpoint.cpp` file, it is

simply the implementation of `cpoint.h`. Here we implement two methods in the `cpoint.h` declaration file.

Step 2: We continue to build the `ctool` class, and in fact this is a class that only contains static methods that support convenient– functions during the game building process.

```h
// ctool.h
#pragma once
#include "cpoint.h"
#include <mutex>
#include <windows.h>
#include <iostream>
using namespace std;

class ctool {
  public:
    static mutex mtx;
    static void ShowConsoleCursor(bool);
    static void GotoXY(int x, int y);
    static void Draw(char*, int, cpoint[], cpoint&);
};
```

```cpp
// ctool.cpp
#include "ctool.h"
mutex ctool::mtx;
void ctool::ShowConsoleCursor(bool showFlag) {
  HANDLE out = GetStdHandle(STD_OUTPUT_HANDLE);
  CONSOLE_CURSOR_INFO cursorInfo;
  GetConsoleCursorInfo(out, &cursorInfo);
  cursorInfo.bVisible = showFlag; // set the cursor visibility
  SetConsoleCursorInfo(out, &cursorInfo);
}
void ctool::GotoXY(int x, int y) {
  COORD crd = { x,y };
  SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE), crd);
}
void ctool::Draw(char* str, int i, cpoint p[], cpoint& _p) {
  mtx.lock();
  _p = p[i];
  ctool::GotoXY(_p.getX(), _p.getY());
  cout << str;
  mtx.unlock();
}
```

Code explanation: In this code we use struct `COORD`, this is a structure for handling coordinates on the console screen. We assign the coordinates and coordinates to the coord variable and then set the position on the screen using the "`SetConsoleCursorPosition`" function. Note: this function needs an object that is the console screen (black screen), so we also need a pointer to this object (`HANDLE` can be seen as a long number). We get it by calling the function "`GetStdHandle`" with the parameter a flag `STD_OUTPUT_HANDLE`. Besides, we have the function "`ShowConsoleCursor`" to help hide the cursor in the console screen (Simply increasing aesthetics). Especially in the "`Draw`" method, we see the use of the `mtx` object (in procedural programming called the `mtx` variable). Because we only have one resource, the "`cout`" object, to "draw" on the screen, but we have two threads, including the "character thread" and the "bullet thread," that we want to use (here, illustrated by so there are only 2 widgets, a real game can have more). Therefore, the `mtx` object helps coordinate the smooth division of resources for 2 threads to use together. Usage is quite simple with two commands "`lock`" and "`unlock`". Note that the "`Draw`" method, after a specified period of time, will be responsible for continuously drawing the point '`_p`' along the path contained in the array '`p`'.

Step 3: Next, we build the `cenemy` class, representing the character that will go through the map. These are the characters that need to be destroyed by towers (`ctower`).

```h
// cenemy.h
#pragma once
#include "cpoint.h"

class cenemy {
```

```cpp
// cenemy.cpp
#include "cenemy.h"

cenemy::cenemy() {
  for (int i = 0; i < cpoint::MAP_SIZE * cpoint::MAP_SIZE;i++)
```

```
  private:
    int _speed;
    cpoint _start, _end, _curr;
    // Array contains the path for the cenemy object from _start to _end
    cpoint _p[cpoint::MAP_SIZE * cpoint::MAP_SIZE];
    // Data for pathfinding for cenemy objects
    int dd[4], dc[4];
  public:
    cenemy();
    cenemy(cpoint tstart, cpoint tend, cpoint tcurr);
    cpoint* getP() { return _p; }
    cpoint getStart() { return _start; }
    cpoint getEnd() { return _end; }
    int getSpeed() { return _speed; }
    cpoint getCurr() { return _curr; }
    void setSpeed(int tspeed) {
      if (tspeed > 0 && tspeed < 10) _speed = tspeed;
    }
    void setStart(cpoint tstart) {
      if (...) _start = tstart;
    }
    void setEnd(cpoint tend) {
      if (...) _end = tend;
    }
    void setCurr(cpoint tcurr) {
      if (...) _curr = tcurr;
    }
    void findPath(cpoint[][cpoint::MAP_SIZE], cpoint, cpoint);
  private:
    void calcPath(int[][cpoint::MAP_SIZE], int, cpoint, cpoint, int=1);
};
```

```
    _p[i] = { 0,0,0 };
  dd[0] = -1; dd[1] = 0; dd[2] = 1; dd[3] = 0;
  dc[0] = 0; dc[1] = -1; dc[2] = 0; dc[3] = 1;
  _start = _end = _curr = { 0,0,0 };
  _speed = 3;
}
cenemy::cenemy(cpoint tstart, cpoint tend, cpoint tcurr) : cenemy() {
  _start = tstart;
  _end = tend;
  _curr = tcurr;
}
void cenemy::calcPath(int a[][cpoint::MAP_SIZE], int n, cpoint s,
cpoint e, int step) {
  a[s.getX()][s.getY()] = step;
  if (s.getX() == e.getX() && s.getY() == e.getY()) {
    int k = 1;
    while (k <= step) {
      for (int i = 0; i < cpoint::MAP_SIZE; i++) {
        for (int j = 0; j < cpoint::MAP_SIZE; j++) {
          if (a[i][j] == k) {
            _p[k - 1] = cpoint::fromRowColToXY({ i,j,0 });
              goto nhan;
          }
        }
      }
      nhan: k++;
    }
    return;
  }
  for (int i = 0; i < 4;i++) {
    int dmoi = dd[i] + s.getX(), cmoi = dc[i] + s.getY();
    if (dmoi>=0 && dmoi<n && cmoi>=0 && cmoi<n && a[dmoi][cmoi] == 0)
    calcPath(a, n, { dmoi, cmoi, 0 }, e, step + 1);
  }
  a[s.getX()][s.getY()] = 0;
}
void cenemy::findPath(cpoint a[][cpoint::MAP_SIZE], cpoint s, cpoint e)
{
  int ta[cpoint::MAP_SIZE][cpoint::MAP_SIZE];
  for (int i = 0; i < cpoint::MAP_SIZE; i++) {
    for (int j = 0; j < cpoint::MAP_SIZE; j++)
      ta[i][j] = a[i][j].getC();
  }
  s = cpoint::fromXYToRowCol(s);
  e = cpoint::fromXYToRowCol(e);
  calcPath(ta, cpoint::MAP_SIZE, s, e);
}
```

Code explanation: The `cenemy` class needs the `_speed` property to represent the character's movement speed (the larger the value, the faster the character moves), `_start`, `_end` and `_curr` to represent the character's starting, ending and current coordinates. This is important information in the calculation process. In particular, the one-dimensional array `_p` will store all the coordinates and characters (`cenemy` objects) must pass through on the map. The two sub-arrays `dd` and `dc` are just additional information that supports the path finding algorithm of the recursive method "`calcPath`". We note that the "`set`..." methods of the `cenemy` class need to check the input before assigning values to the object's properties. In short, in this class we see that the functions "`findPath`" and "`calcPath`" are very important, they will rely on the map (matrix parameter `a`) to calculate the path for the `cenemy` character.

Step 4: The next step we will build the `cbullet` class, this is the class that represents bullets fired by towers, and of course can also be used for `cenemy`. However, in this illustration, the `cenemy` cannot shoot yet.

```
// cbullet.h
#pragma once
#include "cpoint.h"

class cbullet {
  // _n is not the number of bullets, but the current size
```

```
// cbullet.cpp
#include "cbullet.h"

cbullet::cbullet() {
  _n = 0;
  _speed = 4;
```

```
of the bullet path array
  int _n, _speed;
  cpoint _p[cpoint::MAP_SIZE * cpoint::MAP_SIZE];
  // map information is used to calculate the path array _p
  cpoint _m[cpoint::MAP_SIZE][cpoint::MAP_SIZE];
  cpoint _curr; // current position of the bullet
  public:
    cbullet();
    void updateMap(int, int, cpoint);
    cpoint getCurr() { return _curr; }
    void setCurr(cpoint tcurr) {
      if (...) _curr = tcurr;
    }
    cpoint* getP() { return _p; }
    int getSpeed() { return _speed; }
    int getN() { return _n; }
    void setN(int tn) {
      if (...)_n = tn;
    }
    void setSpeed(int tspeed) {
      if (...) _speed = tspeed;
    }
    int queryCFromRowCol(int, int);
    int calcPathBullet(cpoint);
};
```

```
  _curr = { 0,0,0 };
  for (int i = 0; i < cpoint::MAP_SIZE *
cpoint::MAP_SIZE;i++) _p[i] = { 0,0,0 };
  for (int i = 0; i < cpoint::MAP_SIZE;i++) {
    for (int j = 0; j < cpoint::MAP_SIZE;j++) {
      _m[i][j] = { 0, 0, 0 };
    }
  }
}
void cbullet::updateMap(int i, int j, cpoint v) {
  if (...)_m[i][j] = v;
}

int cbullet::queryCFromRowCol(int row, int col) {
  if (...)return -2;
  else {
    for (int i = 0; i < cpoint::MAP_SIZE; i++) {
      for (int j = 0; j < cpoint::MAP_SIZE; j++) {
        cpoint tmp = cpoint::fromXYToRowCol(_m[i][j]);
        if (tmp.getX() == row && tmp.getY() == col) {
          return _m[i][j].getC();
        }
      }
    }
  }
}
int cbullet::calcPathBullet(cpoint tower) {
  cpoint tmp = cpoint::fromXYToRowCol(tower);
  int row = tmp.getX(), col = tmp.getY(), i = 0;
  do {
    col++; row--;
    if (queryCFromRowCol(row, col) == 0) {
      _p[i] = cpoint::fromRowColToXY({ row,col,0 });
      i += 2;
    }
    else break;
  } while (i < cpoint::MAP_SIZE);
  _n = i;
  for (i = 1; i < _n;i += 2) {
    _p[i] = {_p[i - 1].getX() + 2,_p[i - 1].getY() - 1,0};
  }
  _curr = { _p[0].getX(),_p[0].getY(),_p[0].getC() };
  return _n;
}
```

Code explanation: In this code, we note that there is a method "`calcPathBullet`" used to calculate the `_p` path for the bullet. Note that in this illustration we will have a diagonal path with the initial coordinates, we can use the coordinates of the tower (`ctower`). Apparently the "`calcPathBullet`" method uses convenient–functions like "`queryCFromRowCol`" and "`fromRowColToXY`" of `cpoint`. The group of students read it themselves to understand how it works, and of course this method is not good, the group can find ways to improve or replace it because the path of the bullet does not have to be diagonal but can be of any shape. In this class, we note that there is a method "`updateMap`" that actually wants to get map information `_m` for the `cbullet` object to help calculate the path in the "`calcPathBullet`" function earlier. In addition, we also see that `cbullet` also has quite basic properties such as `_speed` (bullet speed), `_curr` (current coordinates of the bullet) and `_n`. In which the property `_n` is simply the number of points in the path array `_p` (Because we are using a static array, we need this extra property to determine the actual number of points in the array `_p`).

Step 5: We build the `ctower` class (this class, because it is only for illustration, will contain **ONE** `cbullet`).

```h
// ctower.h
#pragma once
#include "cbullet.h"

class ctower {
  cpoint _location;
  cbullet _cb;
  public:
    ctower() { _location = { 0,0,0 }; }
    cpoint getLocation() { return _location; }
    void setLocation(cpoint tl) {
      if (...) { _location = tl; }
    }
    void setMapForBullet(cpoint[][cpoint::MAP_SIZE]);
    void calcPathBullet() {
      _cb.calcPathBullet(_location);
    }
    // Returning references is quite dangerous.
    // You need to build test functions for the cbullet
    // class before changing the value of the internal
    // properties.
    cbullet& getBullet() { return _cb; }
};
```

```cpp
// ctower.cpp
#include "ctower.h"

void ctower::setMapForBullet(cpoint a[][cpoint::MAP_SIZE])
{
  for (int i = 0; i < cpoint::MAP_SIZE; i++) {
    for (int j = 0; j < cpoint::MAP_SIZE; j++) {
      _cb.updateMap(i, j, a[i][j]);
    }
  }
}
```

Code explanation: It's easy to see that this class has a `_location` property that represents the coordinates of the tower. In addition, we also have the property `_cb`, a `cbullet` object. We pay attention to the "`calcPathBullet`" method: because the `cbullet` object depends on the position of the `ctower` object, the `ctower` object is responsible for supporting the `cbullet` object in calculating the path. In addition, in this class there is also a method "`setMapForBullet`" that actually supports building a map for `cbullet` _cb. Finally, we see that the "`getBullet`" method returns a reference without a value. Because during the calculation process, we need to change the value inside `cbullet` object of `ctower` object, so we need references (you can use pointer but it will be complicated in syntax). However, inside the `cbullet` class, the "`set`..." methods or methods that change property need to be checked carefully before assigning values.

Step 6: Next, we build `cmap` class. It will contain **ONE** `cenemy` and **ONE** `ctower`.

```h
// cmap.h
#pragma once
#include "ctool.h"
#include "cenemy.h"
#include "ctower.h"

class cmap {
  cenemy _ce;
  ctower _ctw;
  // Map matrix _m
  cpoint _m[cpoint::MAP_SIZE][cpoint::MAP_SIZE];
  public:
    cmap();
    //-1 là tường, 0 là trống, 1 là Tower
    void resetMapData();
    void makeMapData();
    void drawMap();

    cenemy& getEnemy() { return _ce; }
    ctower& getTower() { return _ctw; }
```

```cpp
// cmap.cpp
#include "cmap.h"

cmap::cmap() {
  resetMapData();
  makeMapData();
}
void cmap::resetMapData() {
  for (int i = 0; i < cpoint::MAP_SIZE; i++) {
    for (int j = 0; j < cpoint::MAP_SIZE; j++) {
      _m[i][j] = {4 * j + cpoint::LEFT + 2,2 * i + cpoint::TOP + 1,-1 };
    }
  }
}
void cmap::makeMapData() {
  _m[2][0].setC(0); _m[2][1].setC(0); _m[2][2].setC(0); _m[2][3].setC(0);
  _m[2][7].setC(-2); _m[2][8].setC(-2);
  _m[3][1].setC(-2); _m[3][2].setC(-2); _m[3][3].setC(0);
  _m[3][4].setC(0); _m[3][5].setC(0); _m[3][6].setC(0);
  _m[3][7].setC(-2); _m[3][8].setC(-2);
```

```
};
```

```
  _m[4][1].setC(-2); _m[4][2].setC(-2); _m[4][6].setC(0);
  _m[4][7].setC(0); _m[4][8].setC(0);
  _m[5][8].setC(0);
  _m[5][6].setC(0);
  _m[6][5].setC(0);
  _m[6][8].setC(0);
  _m[7][3].setC(0); _m[7][4].setC(0); _m[7][5].setC(0); _m[7][6].setC(0);
  _m[7][7].setC(0); _m[7][8].setC(0);
  _m[8][1].setC(-2); _m[8][2].setC(-2); _m[8][3].setC(0);
  _m[9][1].setC(-2); _m[9][2].setC(-2); _m[9][3].setC(0);
  // Initializes the current, starting and ending coordinates that
  // character passes through
  _ce.setStart(_m[2][0]);_ce.setEnd(_m[9][3]);_ce.setCurr(_m[2][0]);
  // Calculate the path for the character
  _ce.findPath(_m, _ce.getStart(), _ce.getEnd());
  // Start assigning coordinates for Tower _ctw
  _ctw.setLocation(_m[9][2]);
  // Updated map information for cbullet object of Tower _ctw
  _ctw.setMapForBullet(_m);
  // Calculate the path for the cbullet object of Tower _ctw
  _ctw.calcPathBullet();
}
void cmap::drawMap() {
  // Draw map data _m
  for (int i = 0; i < cpoint::MAP_SIZE; i++) {
    for (int j = 0; j < cpoint::MAP_SIZE; j++) {
      ctool::GotoXY(_m[i][j].getX(), _m[i][j].getY());
      if (_m[i][j].getC() == -1) cout << '+';
      else if (_m[i][j].getC() == -2) cout << 'U';
    }
  }
  // Draw _ctw
  ctool::GotoXY(_ctw.getLocation().getX(), _ctw.getLocation().getY());
  cout << 'T';
}
```

Code explanation: `cmap` class contains virtually all other classes including `cenemy` and `ctower`. Properties of this class, in addition to the 2 objects `cenemy` and `ctower`, are also the matrix `_m` containing map information. Here the "`makeMapData`" method will build map information to know where the wall is (value -1), where the road is (value 0), and where the tower is built (value 1). In this method we also use to set the positions for `cenemy` and `ctower` as well as calculate the path array for them. The 2 functions "`getEnemy`" and "`getTower`" also return references for the same reason explained in step 5.

Step 7: Next, we build the `cgame` class to handle the game's tasks, here it is simply the "`startGame`" function, we also need to build methods like "`exitGame`" or "`pauseGame`" to perform the exit and stop the game when needed.

```
// cgame.h
#pragma once
#include "cmap.h"

class cgame {
  private:
    cmap _map;
    bool _ISEXIT1, _ISEXIT2;
  public:
    cgame();
    bool getIsExist1() { return _ISEXIT1; }
    void setIsExist1(bool b) { _ISEXIT1 = b; }
    bool getIsExist2() { return _ISEXIT2; }
    void setIsExist2(bool b) { _ISEXIT2 = b; }
    cmap& getMap() { return _map; }
    void startGame();
};
```

```
// cgame.cpp
#include "cgame.h"

cgame::cgame() {
  _ISEXIT1 = _ISEXIT2 = false;
}
void cgame::startGame() {
  system("cls");
  _map.drawMap();
}
```

Code explanation: In this class there is only **ONE** `cmap` object (there can be multiple maps representing different levels, depending on the design team). In particular, there are two properties `_ISEXIT1` and `_ISEXIT2`. This is for illustration purposes only, helping to coordinate the two threads to work smoothly (Be explained in the step below). In short, this class is nothing remarkable.

Step 8: Next is the `MainPrg.cpp` which is responsible for handling the entire game.

```
// MainPrg.cpp
#include "cgame.h"
#include <thread>

void ThreadFunc2(cgame&);
void ThreadFunc1(cgame&);

int main() {
  cout << "Press any key to start demo: ";
  cin.get();
  ctool::ShowConsoleCursor(false);
  cgame cg;
  cg.startGame();

  thread t1(ThreadFunc1, std::ref(cg));
  thread t2(ThreadFunc2, std::ref(cg));
  t1.join();
  t2.join();

  ctool::GotoXY(0, 20); // Jump to the last line to exit the program
  return 0;
}

void ThreadFunc2(cgame& cg) {
  int i = 0;
  cenemy& ce = cg.getMap().getEnemy(); // Get the cenemy object
  cpoint _ENEMY;
  cbullet& cb = cg.getMap().getTower().getBullet(); // Get the cbullet object
  int _SPEED = cb.getSpeed(), _NBULLET = cb.getN();
  cpoint* _BULLET1_P = cb.getP();
  cpoint _BULLET1 = cb.getCurr();
  while (!cg.getIsExist1()) {
    ctool::Draw((char*)"o", i, _BULLET1_P, _BULLET1);
    cb.setCurr(_BULLET1);
    _ENEMY = ce.getCurr();
    if (_BULLET1.getX() == _ENEMY.getX() && _BULLET1.getY() == _ENEMY.getY()) {
      ctool::Draw((char*)" ", i, _BULLET1_P, _BULLET1);
      cg.setIsExist1(true);
      break;
    }
    else i++;
    Sleep(1000 / _SPEED);
    ctool::Draw((char*)" ", i - 1, _BULLET1_P, _BULLET1);
    if (i == _NBULLET)i = 0;
  }
  cg.setIsExist2(true);
}

void ThreadFunc1(cgame& cg) {
  int i = 0;
  cenemy& ce = cg.getMap().getEnemy(); // Get the cenemy object
  cpoint _ENEMY = ce.getCurr(), _END = ce.getEnd(); int SPEED = ce.getSpeed();
  cpoint* _P = ce.getP(); // Get the _p path of the cenemy object
  cbullet& cb = cg.getMap().getTower().getBullet(); // Get the cbullet object
  cpoint _BULLET1;
  while (!cg.getIsExist1()) {
    ctool::Draw((char*)"K", i, _P, _ENEMY);
    ce.setCurr(_ENEMY);
    _BULLET1 = cb.getCurr();
    if (_ENEMY.getX() == _END.getX() && _ENEMY.getY() == _END.getY()) {
      break;
    }
    else if (_ENEMY.getX() == _BULLET1.getX() && _ENEMY.getY() == _BULLET1.getY()) {
      ctool::Draw((char*)" ", i, _P, _ENEMY);
```

```
      break;
    }
    else i++;
    Sleep(1000 / SPEED);
    ctool::Draw((char*)" ", i - 1, _P, _ENEMY);
  }
  cg.setIsExist1(true);
}
```

Code explanation: Consider the function "`main`" at first, we ask the user to press any key before playing. Actually, this is a trick to take the time to enlarge our console screen when running in *visual studio*. The student group can freely create other ways. Then we will create an object of type `cgame` named `cg` and call the "`startGame`" method. Next, we create two threads `t1` and `t2` with two functions running in parallel, "`ThreadFunc1`" and "`ThreadFunc2`" with a reference passing mechanism (The student group learns the syntax themselves). We call the "`join`" method from two objects `t1` and `t2` to ensure that the "`main`" function will wait until both threads `t1` and `t2` finish. After the two threads finish, we will end the "`main`" function as shown in the code. Now let's analyze the two functions "`ThreadFunc1`" and "`ThreadFunc2`"

- `ThreadFunc1`: This function is mainly responsible for continuously drawing the character (character '*K*') to the console screen. We see that if `_ENEMY` '*K*' reaches the destination or collides with the `_BULLET1` bullet, the game is considered over. And before ending the function "`ThreadFunc1`", we will set the `_ISEXIT1` flag of the `cg` object to `true`.

- `ThreadFunc2`: This function is mainly responsible for continuously drawing the bullet `_BULLET1` (character '*o*') to the console screen. We see that only when `_BULLET1` collides with `_ENEMY` does the game stop. And before stopping we will set `true` for both `_ISEXIT1` and `_ISEXIT2`.

So, with the above layout, we will have 2 cases where the game ends: when `_ENEMY` '*K*' reaches the destination OR `_ENEMY` '*K*' is hit by `_BULLET1` bullet. Note here that the naming illustration is quite random and confusing. The student group should choose a more systematic naming method to make it easier to maintain the code.

4 GAME CLASS– DIAGRAM

Next, to easily visualize the game design overview, let's look at the image below

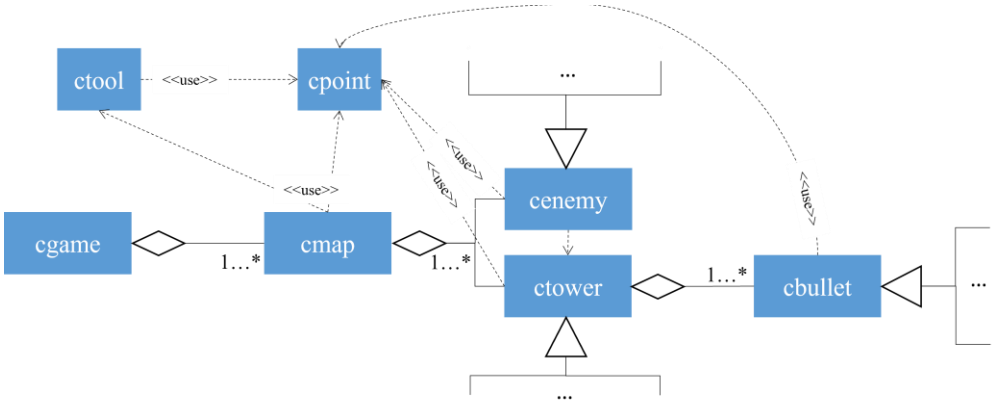

Figure 3: Class–diagram of game

Here we see that the `ctool` class plays the role of a class containing static methods used by most other classes. The illustration design is quite simple, just classes containing each other. For example, the `ctower` class contains the `cbullet` class, the `cmap` class contains `ctower` and `cenemy`, and finally `cgame` will contain a `cmap`. Besides, when developing the game, there may be many types of bullets, many types of characters and many different types of towers. Therefore, we can design inherited subclasses as shown in the illustration above. We note that in the illustration there are no basic functions such as save/load or other advanced functions. Therefore, the actual class diagram may be much more complicated.

5 REQUIREMENTS

In the above instructions, we still lack basic functions

5.1 Handle collisions between characters and bullets

A group of students should design collision effects to make the game more attractive

5.2 Handles saving games/loading saved games

Students add the save/load function when the user needs to save the game or select a previously saved game to continue playing.

5.3 Design how to play when the game screen appears

In the instructions, when running the program, the game immediately starts. However, the student group should **redesign** it so that when entering the game, the player must be shown the map, the location where towers can be built or the number of characters/paths so that the player can predict and choose appropriate parameters suitable to win the level.

5.4 Handling of game levels

Students build the game having **at least 4 levels** and corresponding to 4 different maps in the game. Speed can be used as one of the difficulty factors for each level.

5.5 Building menu

Build game menus like “new game”, “load game”, or “setting…”. It is necessary to build 2 types of sounds: background sound and playing sound. Remember to have the on/off function.

5.6 Other advanced functions

In addition to the mentioned functions, groups need to think of a special function of their own (Note that this is a special function, groups do not share ideas with other groups)